AD

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/445, 9/45

A1

(11) International Publication Number: WO 96/12224

(43) International Publication Date: 25 April 1996 (25.04.96)

(21) International Application Number: PCT/US95/06059

(22) International Filing Date: 12 May 1995 (12.05.95)

(30) Priority Data:
08/324,810 18 October 1994 (18.10.94) US

(71) Applicant: MARCAM CORPORATION [US/US]; 95 Wells Avenue, Newton, MA 02159 (US).

(72) Inventors: REED, Harvey, G.; 7 Royal Crest Drive #1, Marlborough, MA 01752 (US). DALTON, John, T.; 166 Hunt Road, Chelmsford, MA 01824 (US). AIBEL, Jonathan, B.; 574 Harrington Avenue, Concord, MA 01752 (US). SCIANDRA, Stephen, A.; 34 Newcombs Mill Road, Kingston, MA 02364 (US).

(74) Agents: POWSNER, David, J. et al.; Choate, Hall & Stewart, Exchange Place, 53 State Street, Boston, MA 02109-2891 (US).

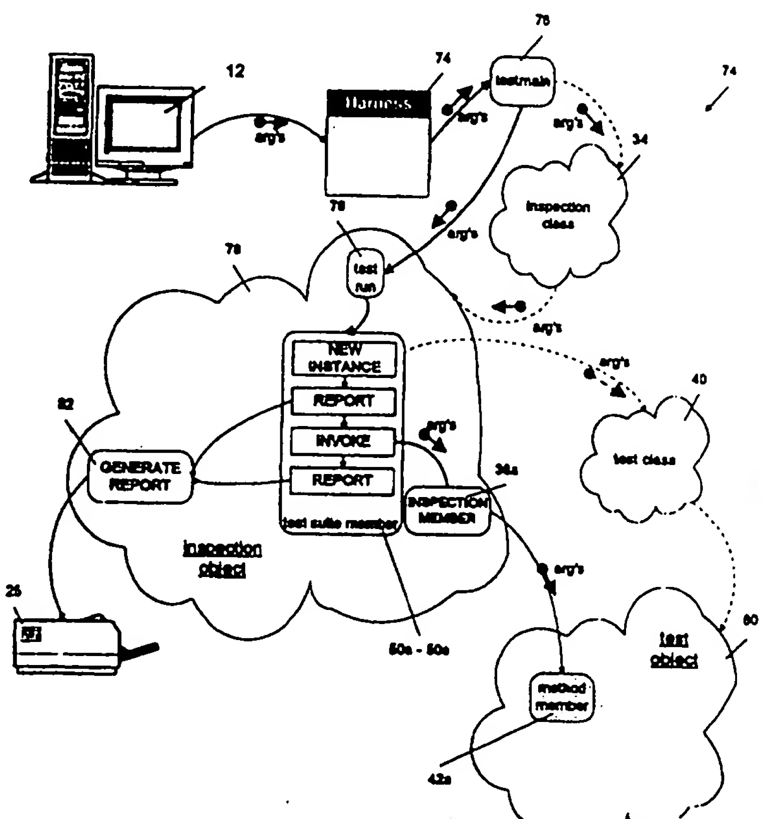
(81) Designated States: CA, JP, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.

(54) Title: METHOD AND APPARATUS FOR TESTING OBJECT-ORIENTED PROGRAMMING CONSTRUCTS

(57) Abstract

The invention provides methods and apparatus for generating, from a source signal defining a subject class to be tested (40), an inspection signal defining an inspection class (34) that has one or more members for: (i) generating a test object as an instantiation of the subject class or a class derived therefrom, (ii) invoking one or more of the selected method members of the test object, and (iii) generating a reporting signal based on an outcome of invocation of those members. The inspection class, as defined by the inspection signal, can include one or more method members (50f), referred to as "inspection members", for testing corresponding method members of the test object (and, therefore, in the subject class). So-called "test suite members" (50a-50e), that are also defined as part of the inspection class, exercise the inspection members. The invention also provides methods and apparatus for responding to an inspection signal to create an inspection object instantiating the inspection class. Members of the inspection object are invoked to create the test object (80), to invoke method members thereof and to generate a signal reporting an effect of such invocation. The test object members can be invoked by corresponding inspection members of the inspection class, which in turn can be invoked by test suite members of the inspection class.



METHOD AND APPARATUS FOR TESTING OBJECT-ORIENTED PROGRAMMING CONSTRUCTS

Reservation of Copyright

BACKGROUND OF THE INVENTION

The invention described herein pertains to digital data processing and, more particularly, to methods and apparatus for testing object-oriented programming constructs.

There are many competing factors determining how an organization tests its software. In the commercial environment, this includes allocating development resources, motivating developers to write unit tests, and maintaining those tests over the lifetime of the product. The central problem is focusing limited development resources on the unique aspects of unit tests, minimizing the overhead of creating test environments and packaging. Secondary factors include choosing which techniques (state-based, boundary analysis, etc.) are applicable to validate each production unit.

Traditional unit testing validates the function points within a software product to verify the correctness of its operation. Often, the strategy is to write a dedicated test program for each unit, requiring separate test control and execution, set up of the test environment, and production of output to describe the execution of the test, as shown in **FIGURE 1**.

Implementing this strategy is often informal and ad hoc. Frequently unit testing is abandoned because of the difficulty of this approach. When a developer determines that some code does need unit testing, he will create his own test harness, writing code to set up the test environment, run the test case, output the results and shutdown the test environment. The developer is concerned about all support issues, such as memory management, output device control and formatting, and initializing and cleaning up the application-specific environment. The actual test code is generally a small percentage of the entire unit test.

This practice also tends to be invasive, where conditional test code is added to the production code to aid in debugging. Typically this code consists of commands to output the value of variables, trace the execution of the code and perhaps validate the state of the

Current research into the testing of object-oriented programs is concerned with a number of issues. The black-box nature of classes and objects implies that a black-box testing technique would be beneficial. A number of black-box approaches have been developed, including the module validation technique of Hoffman *et al* "Graph-Based Class Testing," Proceedings of the 7th Australian Software Engineering Conference, ASWEC (1993) and "Graph-Based Module Testing," Proceedings of the 16th Australian Computer Science Conference, pp. 479 - 487, Australian Computer Science Communications, Queensland University of Technology, and the ASTOOT suite of tools by Frankl *et al*, "Testing Object-Oriented Programs," Proceedings of the 8th Pacific Northwest Conference on Software Quality, pp. 309 - 324 (1990). Both of these techniques concentrate upon the modular nature of object oriented programs. However Fielder, "Object-Oriented Unit Testing," Hewlett-Packard Journal, pp. 69 74, April (1989) notes that a more adequate testing technique can be gained by combining black-box testing with the use of a whitebox coverage measure.

In view of the foregoing, an object of the invention is to provide improved methods and apparatus for testing object-oriented programming constructs. More particularly, an object is to provide such methods and apparatus for testing classes in object-oriented programs and libraries.

A related object is to provide such methods and apparatus for testing all members of a class and exercising each member over a full range of expected runtime parameters.

Still another object is to provide such methods and apparatus for implementation in a wide range of digital data processing operating environments.

SUMMARY OF THE INVENTION

The invention provides methods and apparatus for testing object-oriented software systems and, more particularly, class constructs that provide the basis for programming and data structures used in those systems.

In one broad aspect, a method according to the invention calls for generating, from a source signal defining a subject class to be tested, an inspection signal defining an inspection class that has one or more members for (i) generating a test object as an instantiation of the subject class or a class derived therefrom, (ii) invoking one or more selected method members of the test object, and (iii) generating a reporting signal based upon an outcome of invocation of those members. The source and inspection signals can be, for example, digital representations of source-code programming instructions of the type typically contained in source-code "header" files.

selected member methods. Those arguments can, by way of example, be specified interactively or generated automatically.

In a related aspect of the invention, invocation of a member function is reported by comparing the results of invocation of a member function with an expected value. In the case of disagreement, an error message can be displayed, e.g., to the user console or to an output file. By way of further example, the contents of the test object can be "dumped" following invocation. Moreover, signals indicative of the results of these and other results of invocation, e.g., statistics regarding the number of successful and erred member function calls, can be stored in data members of the inspection object. Reporting signals, such as the foregoing, generated in accord with the invention can be generated at selected verbosity levels.

Still other aspects of the invention provide an apparatus for testing object-oriented software systems having functionality for carrying out the functions described above.

These and other aspects of the invention are evident in the description that follows and in the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

A more complete understanding of the invention may be attained by reference to the drawings, in which:

Figure 1 depicts a prior art strategy for unit testing of procedural code;

Figure 2 depicts a prior art strategy for testing object-oriented code;

Figure 3 depicts a preferred computing system for use in practice of the invention;

Figure 4 depicts a production class and an inspection class according to the invention for testing thereof;

Figure 5 depicts a relationship between the inspection class and test monitor and test kernel superclasses, as well as between the test subclass, the production class and the production superclass;

Figure 6 depicts a test suite and inspection methods comprising an inspection class according to the invention;

Figure 7 depicts illustrative states an object can enter under the persistence paradigm;

commercially operating system tools in connection with libraries constructed in accord with the teachings herein.

As shown in the drawing, method member I_Class::A takes arguments and returns values identical to those of corresponding member A. Member I_Class::A takes, as an additional argument, a pointer to a test object created from the production class or a test class derived therefrom. As further shown in the drawing, method I_Class::A includes instructions for invoking member A of the test object, as well as for reporting invocation, arguments to, and return values from that member. Thus, by way of example, where member A is declared as follows:

```
bool A(Int      arg1);
```

I_Class::A can be declared as follows:

```
bool
I_CLASS::Class:: A(P_Class*  Obj,
                   Int       arg1)
{
  AnnounceStart("A");              // report start of test
  AnnounceParameter(arg1);         // report parameters that will be
                                   // passed to member to be tested
  bool tstVal = Obj->A(arg1);      // invoke member
  AnnounceRtnValue(tstVal);        // report member result
  AnnounceEnd("A");                // report end of test
  return(tstval);                  // return member result
}
```

Each of the inspection methods take the object under test as a parameter, as well as the rest of the parameters necessary for the production method, as shown in **FIGURE 4**. This not only reduces cut and paste code, but also eliminates the need for specialized test code in the production class that is used only during unit testing. It is much preferred to have instrumentation outside of production code rather than compiled in, even if the compilation is conditional.

The inspection class methods are capable of running against code built for general release as well as development and unit testing. With a "dump" method enabled during development, unit tests facilitated by the inspection class provide a white-box testing environment, allowing the developer to be most effective in determining corrective actions for failed tests. Black-box test techniques are in effect when the dump method is disabled for

Given a set of inspection methods 36a - 36c grouped into an inspection class 34, common services of test output and test environment maintenance are provided by test instrumentation base classes 46, 48. The kernel test class 48 allows infrastructure objects to be tested in isolation. It provides services for output verbosity control, output formatting, memory management leak detection and reporting, expression checking, and reporting of errors encountered.

The monitor test class 46, derived from the kernel test class 48, provides additional services to application classes such as application environment initialization and cleanup. test case setup and cleanup, gross level coverage measurement and reporting. and unit test execution.

ORGANIZATION OF TEST SUITES

Source code and header files providing definitions of the inspection class 34 and test subclass 40 are preferably generated automatically from a header file defining the production class. This results in a reduction in the amount of code the developer needs to write in order to insure thorough testing of the production class. This section examines the definition and interrelationship of unit tests within the inspection class.

Unit tests must achieve a certain minimum standard of coverage. This is ensured by dividing the set of production class methods, e.g., 32a - 32c, into groups according to behavior and testing technique, taking into account dependencies between various groups. For example, all methods depend on the constructor methods. Referring to **FIGURE 6**, production class 30 methods are tested by the following groups of inspection class tests: memory lifecycle 50a, persistence lifecycle 50b, get and set attributes 50c, operator members 50d, and semantics 50e.

Memory lifecycle tests 50a exercise those methods of the production class which affect the life of the corresponding object in memory. These include the constructors, copy constructors, destructor and assignment methods of the production class. Generally, errors which occur in the lifecycle are memory leaks and improper initialization. Lifecycle tests involve creating test objects (based on the production class, or the test class derived therefrom) using each constructor in the production class 30, assigning objects (e.g., in the case of copy constructors), and deleting the test objects from memory. Checks are then made for leftover allocated memory.

Persistence lifecycle tests 50b exercise those objects that are stored in a database (e.g., on disk 22) for later retrieval or storage across invocations of the product. A state-based testing approach is used for persistent methods. The states defined in the memory and

Durham to address some of the problems in adapting testing techniques to object-oriented programming, and is described in Turner *et al*, "The Testing of Object-Oriented Programs," Technical Report TR-13/92, University of Durham, England (1992).

An object's behavior is defined by the code within its operations. Events and situations occur that have to be communicated to the other operations of the object. This communication is usually performed by storing specific values within the attributes of the class. State-based testing models these interactions as a finite-state-automata which is then used to predict the response of each operation to a particular state.

Test cases are generated to validate that at no point in an object' s lifetime can it be placed into an undefined or incorrect state. All operations are validated for the state changes which they are defined for, operations can also be validated for their robustness when faced with an illegal state situation.

An example deals with persistent classes. Persistence is implemented by a root persistent class from which all persistent objects are derived. The persistence paradigm requires six states that an object can enter:

New - the object was created and exists in memory only

Saved - the object exists in the database and not in memory

Open for Read/Write - the object exists (and is identical) in memory and the database and the client may save changes to the object

Modified Read/Write - the object exists in memory and the database with different values and the client may make changes to the object

Open for Read Only - the object exists (and is identical) in memory and the database and the client may not make changes to the object

Deleted - the object does not exist in memory or the database

FIGURE 7 illustrates some of the legal transitions between these states. Thus, for example, an object may transition from a New state 52a, or a Modified Read/Write state 52e, into a Saved state 52b. From that state 52b, an object may transition to an Open Read Only 52c or an Open Read/Write state 52d, whence it may transition to a Deleted state 52f. An object may also transition into a Deleted state from the Modified Read/Write state 52e.

Since the root persistent class is inherited by all persistent classes, these classes share the above states 52a - 52f, although the implementation is made specific to each class by

prototypes corresponding to those in the file 62b. Those skilled in the art will appreciate that the teachings herein are likewise applicable to other programming languages including object-oriented constructs, e.g., Turbo Pascal. Likewise, it will be appreciated that the information in the files may be input or output in other formats (e.g., object code) known in the art.

The statements contained in files 62a, 62b define the inspection class 34 to include members that (i) generate a test object as an instantiation of the test class, (ii) invoke selected method members of the test object, and (iii) generate a reporting signal based upon an outcome of invocation of those members.

The inspection class 34 particularly includes inspection members (also referred to as "inspection" members) that test corresponding method members of the test object. Thus, for example, the inspection class 34 includes inspection members, e.g., 36a - 36c, corresponding to, and taking similar arguments to, constructors, destructors, operators and other method members, e.g., 32a - 32c, in the test object. To facilitate testing, the inspection members (apart from constructors and destructors) have function names similar to those of the corresponding method members of the test object that they test.

The inspection class 34 defined by files 62a, 62b also includes test suite members that invoke or exercise the inspection members for test purposes. The test suite members 50a - 50e that test accessor, transformer, operator or semantically unique members of the test object, as well as the persistence of the test object and memory leaks associated with its creation or destruction. A "test main" member 50f invokes the test suite. The inspection class provides, via inheritance from the test monitor class and test kernel class 48, members 50f that permit uniform tracking and reporting of test coverage and errors.

The statements in files 62a, 62b also define a test class 40 that derives from the production class 30 and, therefore, inherits members, e.g., 32a - 32c, therefrom. The test class 40 duplicates pure virtual functions of the subject class 30, absent programming constructs that denote those functions as having both pure and virtual attributes (e.g., the "pure" and "virtual" keywords). The test class 40 affords the inspection class members, e.g., 36a - 36c, access to members, e.g., 32a - 32c, of the subject class, e.g., via C++ friend declarations.

Generation of the inspection class header and source files 62a, 62b by the ICG 58 may be understood by reference to the Appendix A, providing a listing of a sample production class header file 60 named "utg.hh"; Appendix B, providing a listing of a sample inspection class header file 62a named "i_utg.hh" that is generated by the ICG 58 from file 60; and,

9. The start of the public member functions (and member data) are signaled by the "public" keyword. The keywords "protected" or "private," or the close of the class declaration "};" end the public section. C++ permits multiple public sections within a class declaration.

10. ICG 58 ignores enumerations, indicated by the keyword "enum."

11. Constructors. C++ permits more than one per class. C++ also permits constructors to be overloaded.

12. Copy Constructor. If provided, there is only one. A copy constructor has as its first required argument a reference (&) to an instance of this class (usually declared const). In a preferred embodiment, there is a second, optional argument, a memory manager defaulted to NULL. All classes must have one, but they can be private and unimplemented.

13. Destructor. The definition of most classes include a public destructor.

14. Assignment operator. All classes must have one, however, they can be declared private and not implemented.

15. Conversion operator. This operator, also known as cast, is provided in addition to the operators defined by C++. This operator returns an object of the type identified in the operator statement (e.g., dtULong). Also notice, this member function is const, which is not used in generating the signature of the inspection member function.

16. The addition assignment operator returns a reference to an object of this class type.

17. A static member function (sometimes called a class function) returning an erStatus object. A pointer to a production object is an output parameter. The static specifier is not included in the corresponding statement in the inspection class files 62a, 62b.

18. Public member data is ignored when generating the inspection class files 62a, 62b.

19. The start of the protected member functions (and member data) is signaled by the "protected" keyword. The keywords "public" or "private," or the close of the class declaration "};" ends the protected section. C++ permits multiple protected sections within a class declaration.

20. An overloaded index operator implemented as a const member function. Again, note that the const is ignored for the generated inspection function.

4. The inspection base class declaration is included by the statement #include "idt-test.hh".

5. The production class header is included by the statement #include "utg.hh".

6. Inspection class comment. The !AUTHOR, !REVIEWER, and !REVIEW_DATE keywords and values are defaulted from a template (values from production class header are not used).

7. The inspection class comment keyword !LIBRARY value adds an "I_" prefix to the value from the production class comment (I_ER).

8. The inspection class comment keyword !NAME value adds a "i_" prefix to the name of the class (i_utg).

9. The inspection class generator 58 places a comment indicating that this file 62a is generated by a tool, to wit, the generator 58. The comment is entered in the !TEXT section of the class comment, and includes a date, the name of the tool, and the version of the tool.

10. The test class declaration publicly inherits from its production class to provide access to protected member functions. The test class name has "s_" prepended to the production class name. All member functions are declared in the "public:" section of the test class.

11. The test class declares the inspection class as a "friend." This allows the inspection class to invoke protected (and public) member functions on the production class.

12. Test class constructors and copy constructors are declared and implemented inline to replicate the public and protected interface on the production class. The private interface of the production class is not used.

13. Declaration to satisfy the compiler for a pure virtual member function declared in the production class. The declaration has lost its pure specifier (=0), but not its const function specifier. Also, the virtual specifier from the production class is also not needed (if it is easy to strip off, do it, otherwise the virtual specifier can stay). A comment is generated as an indication to the unit test developer that an inline implementation may be required (if linking causes this function to be unresolved, i.e., it is being called by some object).

14. An enum named FUNCS contains entries for each inspection function and a final entry named TOTAL_FUNCS. Each entry is composed of a prefix of f_ followed by the name of the inspection function. Where functions are overloaded (e.g., multiple constructors) it is necessary to append a number indicating which of the functions this enumeration applies. This enum is used for calculating base coverage analysis.

24. Assignment operator inspection function. The first argument, as a general rule for all member functions, is a pointer to an instance of the test class, followed by the arguments from the production class assignment operator. The return type is modified to return a test class (s_utg&) instead of a production class (utg&). The suggested name for this operator is oper_equal.

25. The conversion operator must return an object of the same type as the conversion. Since a conversion operator itself has no arguments, the only argument becomes a pointer to an instance of the test class. The suggested naming for conversion operators is oper_<conversion-type>.

26. The addition assignment operator modifies the return type to a test class, and inserts a pointer to an instance of the test class, followed by the arguments defined by the production class. The suggested naming for this operator is oper_plusequal.

27. The static production member function has lost its static specifier in the inspection class. Also, production class parameter type has been replaced with the test class.

28. The index operator follows the same general rules as described for the assignment operator. The suggested name is oper_index.

29. The virtual setID member function contains a default argument, which is a constant pointer to a constant object. The virtual specifier from the production class is not needed on the inspection class (if it is easy to strip off then do it, otherwise the virtual specifier can stay). A pointer to an instance of the test class is inserted as the first argument, with the production class arguments following; including the default argument (=(const dtChar* const) NULL).

30. The start of the "private" section, and the end of the "public" section.

31. Declaration of the inspection class' copy constructor and assignment operator member functions. These are not implemented in the inspection class source file 62b.

32. Nonclass functions are declared outside of the inspection class. Their return type and arguments are preserved (a pointer to an idtTest object is inserted as the first argument, and production class types are modified to test class types). The name is changed so it does not conflict with the production header nonclass function name. A suggested naming scheme for nonclass functions is <inspection-class-name>_<nonclass-function-name>. For example, export becomes i_utg_export.

14. Invoke newInstance using mm, substitute test class name. In further embodiments, a constructor/destructor test (TLIFE_START through TLIFE_END) can be generated for each existing production class constructor. Likewise, they may include argument types in where /*PARAMETERS,*/ currently appears.

15. Copy as is.

16. Invoke newInstance using mm, substitute test class name.

17. Invoke copyInstance using mm2, substitute test class name. In further embodiments, a constructor/destructor test (TLIFE_START through TLIFE_END) can be generated for each constructor/copy constructor combination. Likewise, they may include argument types in where /*PARAMETERS,*/ currently appears.

18. Copy as is.

19. Invoke newInstance using mm, substitute test class name.

20. Invoke newInstance using mm2, substitute test class name. In further embodiments, a constructor/destructor test (TLIFE_START through TLIFE_END) can be generated for each constructor/assignment operator combination. Likewise, they may include argument types in where /*PARAMETERS,*/ currently appears.

21. Copy as is.

22. Three invocations of newInstance using mm, mm2, and mm. Substitute test class name.

23. Copy as is.

24. Invoke newInstance using mm, substitute test class name.

25. Copy as is.

26. Operators testing comment, copy as is.

27. Operators testing body skeleton, substitute inspection class name for scope name (i_utg::t_Operators).

28. Comment and body for set and query tests. Same rules as requirement 27.

29. Semantics testing comments and body. Same rules as requirement 27.

45. Another version of requirement 43, this time a testVal is being returned.

46. Nonclass functions have a pointer to an idtTest passed in as the first argument. This pointer is used for invoking the announceXxxx member functions. This is done because nonclass functions are not member functions on an inspection object (an inspection object is-a idtTest object).

Generation of the inspection class header and source files 62a, 62b by the ICG 58 may be further understood by reference to the Appendix D, providing a listing of sample production class header 60 named "basresc.hh"; Appendix E, providing a listing of a sample inspection class header file 62a named "i_basres.hh" that is generated by the ICG 58 from the file basresc.hh; and, Appendix F, providing a listing of a sample inspection class code file 62b named i_basres.cc that is also generated by the ICG 58 from the file basresc.hh.

The code generation operations revealed Appendices D - F include support for basic persistence testing of a persistable (or persistent) object is the body using the "t_persist()" function and two additional support member functions for setting up valid user keys and persistent objects.

The items listed below refer to the annotations made on the listings in Appendices D - F. In the description of each item, reference is made to the three classes involved in the unit test:

1. The source production class 30, which is a concrete persistent class.

2. The test class 40. This is a generated typedef which provides access to the public member functions of the production persistent class.

3. The inspection class 34. This class performs tests on instances of the test class 40.

As above, where dictated by context, references in the explanatory items below to the "production class," the "inspection class" and the "test class" shall be interpreted references to the corresponding files 60, 62a, 62b.

Explanation of Annotations to Production Class
File 60 Listing of Appendix D

1. User Key class name. A user key is a class that contains text which is the name of a persistent object. This can be obtained from scanning the second parameter type of the openForChange(), openForReview(), and saveAs() member functions. In an alternate embodiment, a hint as to the icg is obtained from the production class developer via a !USERKEY keyword.

9. The qryCoverage member function, which is currently being generated.

10. The user key setup member function body, setupUKs(). This is copied as is, except where noted in 11.

11. User key arguments for setupUKs() requires substitution of the user key class name.

12. The persistent object setup member function body, setupObj(). This is copied as is, except where noted in 13.

13. Test class pointer argument for setupObj() requires substitution of the test class name.

UNIT TEST HARNESS

With continued reference to **FIGURE 9**, the inspection class source file 62b generated by ICG 58 is compiled, linked and loaded (see element 64), along with production class header and source files 60, 68, as well as library files containing the test monitor class definitions 70 and the test kernel class definitions 72. Compiling, linking and loading is performed in a conventional manner, e.g., using a C++ compiler sold by Microsoft Corporation. The resulting file 66 containing executable code for the unit test harness is executed on digital data processor 12 to reconfigure that processor as the unit test harness.

FIGURE 10 depicts operation of the unit test harness 74. Particularly, testmain() routine 76 creates an inspection object 78 as an instantiation of the inspection class 34 defined in files 62a and 62b. Testmain() also invokes testrun(). The testrun() method 79 of the inspection object invokes test suite members 50a - 50e of the inspection object, each of which (i) creates a test object 80 instantiating the test class 40 (defined in file 62b), (ii) invokes a members of the test class, e.g., 42a, via corresponding inspection members, e.g., 36a, of the inspection class, and (iii) utilizes reporting members 82 of the inspection object 78 to report results of the invocation of the test class members. Reports can be generated to printer 25, to monitor 24, to a file on disk drive 22, or in other media as known in the art.

As illustrated, the test harness 74 accepts arguments, e.g., from the user, for application to the test object upon or upon invocation of the selected member methods, e.g., 42a. Such arguments can, alternatively, be generated automatically be the test harness 74.

Reports generated by members 82 of the inspection object include those resulting from comparing the results of invocation of the test object member, e.g., 42a, with expected result values. In the case of disagreement, report members 82 cause an error message to be displayed, e.g., to the printer, monitor or a disk file. Report members 82 also permit the data members of the test object 80 to be "dumped" following invocation. The report members 82

T_INIT(methodname) - Initialize a t_* method, setting up a test memory manager, mm

T_CLEANUP(methodname) - Clean up a t_* method, checking the memory manager for leaks.

TEST_START(testname) - Start a test scenario

TEST_END - End a test scenario; announces whether the scenario passed or failed.

TEST(testname, expression) - A combination of TEST_START() and TEST_END for tests which consist of a single expression to check.

checkExcept(expression, id, testname) - Verify that evaluating the expression throws an exception with the id indicated. *CheckExcept* is implemented in systems that do not otherwise support exception handling by the use of the *setjump* and *longjump* ANSI functions. Prior to evaluating the expression, *setjump* is performed to save the current stack frame. If an exception is thrown and it is required to restore the stackframe, *longjump* is performed.

In addition to the foregoing, the inspection object inherits from *idtTest* the testrun() method and other code for running the t_* methods and the code for analyzing the results of the unit tests.

DEVELOPING UNIT TESTS

The sections that follow provide still further discussion of the operation of the inspection class generator 58 and unit test harness 74. Also discussed is designer modification of the inspection class header and source files 62a, 62b for completing test suites for the test harness.

Although inspection class generator 58 creates most of the unit test code 66 from the production class header file 60, the scope of testing may necessitate that the developer complete implementation of the test and inspection classes, as well as the test scenarios themselves. As noted above, in order to manage the potentially large number of test scenarios, the methods are divided into the following logical groups for testing:
LifeCycle - creation and destruction of the object (tests for memory leaks)
Set & Query - Accessors and transformers of the object
Operators - Operator methods
Persist - methods involved in object persistence
Semantics - methods unique to the class.

Note that the destructor is tested along with the constructor in these cases. The basic test is illustrated in Appendix G.

Also note the use of the test class, s_erMsg, in this example. The test class should be used to represent the production class in all unit test code.

Another consideration in constructor operation is their behavior under error conditions. Usually this means testing constructors when there is insufficient memory for the new object.

In one embodiment, out-of-memory exceptions are thrown by the memory manager, and caught at the application level, so object constructors are not involved. However, constructors may require parameter testing.

Parameter testing should be performed when a parameter's value has some restrictions which are enforced by the constructor. In this case, use equivalence partitioning and boundary value analysis to determine the necessary test cases. For example, if an object had a member which was an unsigned integer, but the only legal values were in the range 1 - 100. In this case, test cases need to include calling the constructor with the values 0 (invalid), 1, 50, 100, 101 (invalid) and 500 (invalid) might be used.

If the parameters are pointers, test cases should be written to check behavior when the parameter is NULL. A test case to check the default constructor needs to be written as well.

Copy Constructor Testing

Testing the copy constructor follows the form of tests for the regular constructors, with slight modifications. In particular, the constructed object should be in a different memory area than the copied object to ensure that the copy is completely independent of the original. Referring to Appendix H, the scenario is as follows:

1. construct an object (obj1)with memory manager mm
2. construct the test object, by copying (obj2)with memory manager mm2
3. check the equality of the objects (if an equality operator exists) (obj1==obj2)
4. delete the original object (obj1)
5. announce the test object (obj2)
6. delete the test object (obj2)

Note that deleting the original object catches errors involved with incomplete copying. TLIFE_END check both memory managers for leaks. The copy constructor needs to have a test case for each constructor.

Testing Set and Query Methods — t_SetQrys

The purpose of t_SetQrys is to test the operation of the accessors and transformers of the class; this includes the Query, Set and Validate methods.

Query Methods Testing

The query methods should be tested first, because they can then be used to test the Set methods. There are two approaches to testing the Query methods. If the method returns an unaltered value, the approach is to create an instance of the class with a known value in the member(s) being queried, and then check that the query returns these values, as illustrated in Appendix I.

Note the use of the TEST() macro; the TEST() macro combines TEST_START(), checkExpr() and TEST_END macros, allowing a test which consists of a single expression to be tested in a single line. Since the example test case consists of the expression, "fdMsgTest3 == qryMsgID(msg1)", the test can be implemented as a call to this macro.

The number of test cases that are needed to validate a query method depends on the complexity of the query and boundary conditions. In some queries, the member being queried may not be present; remember to test both the positive (member present) and the negative (member absent) cases, as illustrated in Appendix J. Null and 0 are the most common boundary values; always consider whether test cases for these need to be written.

Another consideration occurs when the value being queried undergoes some transformation either at creation or during the query. In this case, additional test cases would need to check the special cases, assumptions, or boundaries that the calculation imposed on the data. For example, a date class which stored its value in terms of month, day, year, but included a method qryDayOfYear() which calculates the number of days since the start of the year. This method adds to the class of valid test data the conditions of leap and non-leap years (and skipped leap years at century boundaries), and the special case of the year 1752; test cases need include February 29 and December 31, for both a leap and non-leap years, dates in the skipped leap years, such as 1900, and dates in 1752 (depending on locale).

There are no restrictions in setting up test data which satisfies several query unit tests at once, as shown in Appendix J.

```
operator dtULong() const;
```

This is tested in i_erMsg as follows:

```
//
// Test 1: Cast to dtULong
//
const dtULong test1value = 1;

s_erMsg* msg1 = newInstance( test1value, ERMSGTEST_NAME, ERMSGTEST_5,
__FILE__, __LINE__, mm );

        // Do cast and check the value
TEST( "Cast to dtULong", test1value == oper_dtULong( msg1 ) );

deleteInstance( msg1 );
```

Testing Object Persistence — t_Persist

The purpose of t_Persist is to test the object persistence for the class: this includes the Persist and Fetch methods.

Testing Unique Methods — t_Semantics

The purpose of t_Semantics is to test those methods unique to the class.

Test Execution Code

The inspection class generator 58 creates the code which allows the single unit test harness 74 to run the tests. The harness 74 collects arguments from the user to run the test Then, the test harness 74 calls two exported routines: *testname()* and *testmain()*, to figure out what is being tested and to execute the tests. *testname()* returns a dtChar * which names the inspection class. *testmain()* serves as the top level of test execution, calling testrun() which, in turn, calls inspection class methods which test the production class. In the inspection class, two macros are used, *TESTRUNdeclare* and *TESTRUN* (supplied in idtTest 46), so that *testmain* is not coded directly. Implementation of these routines is illustrated in Appendix K in which the class "erMsgi" refers to the class "i_erMsg."

Once the inspection class code has been written, it is built into a dynamic link library, or "DLL." Each inspection class must have its own DLL, so that the *testname()* and *testrun()* routines do not clash.

Verbosity	Class	Sample Messages
SKIP	Skip particular test suite	SKIPPING: t_LifeCycle, verbosity set to SKIP
OFF	None	
LOW	Error messages	ERROR: in test FOO, file FILE, line LINE, bad expression: EXPR
MED	Status messages	Starting test FOO PASS: FOO
HIGH	Object dumps, other debug messages	Current value of mDate: FOO

If the user sets the verbosity level to OFF, the unit tests will not output any messages, but if they set the verbosity level to MED, they will get the error and status messages in the output log file. Appendices L, M and N depict of test runs at the various verbosity levels. SKIP allows execution of a test suite to be skipped, e.g., for purposes of initial test development and debugging.

In addition, the inspection class appends a simple analysis of the run at the end of a unit test execution. testmain() always returns the number of errors detected, so the test harness can display this information even with the verbosity level set to OFF. At LOW and MED verbosity levels, the unit test code 74 reports on the number of tests run and the number of errors found. At HIGH, coverage information is output; this allows the user to determine whether all of the production methods were tested.

Unit Test Output Files

Output by harness 74 and, particularly, by reporting members 82 is placed into five files:

Log file: <file>.out (<file> represents the destination file field in the unit test harness)
Error file: <file>.err
Memory Log file: memory.log
Exception Log file: except.log
Database Error file: sqlerror.log

The log file is the main file for unit test output. All of the output generated by the unit test code is placed into this file, giving the user a place where they can read the messages in context. As described above, the contents of the log file is controlled by setting the verbosity level.

The error file separates the error messages from the rest of the output for easier management. It is not affected by the verbosity level; it always contains all of the error

UTG.HH

3

```
///////////////////////////////////////////////////////////////////////////
//
// FILE_NAME: utg.hh
//
// Copyright 1993 by Marcam Corp., Newton, MA USA
//
// This unpublished copyrighted work contains
// TRADE SECRET information of Marcam Corporation.
//
// Use, transfer, disclosure, or copying without
// its expressed written permission is strictly
// forbidden.
//
//
///////////////////////////////////////////////////////////////////////////
```

1

```
#ifndef utgHH
#define utgHH
```

2

```
#include "dtcore.hh"          // For dtCore
```

4

```
///////////////////////////////////////////////////////////////////////////
// !CLASS_DECL_S
// !LIBRARY EP
// !NAME utg
// !TEXT
// Test case for the unit test generator.  This class header attempts
// to test a number of different items that need to be transformed
// when creating the inspection class header.  This includes the
// public and protected member functions, operators, static and nonstatic
// member functions, and nonclass functions appearing in a header file.
// Also, various styles of using spaces in a member function delcaration
// are tested.
//
// !AUTHOR        John Dalton           (dalton@opl.com)
// !REVIEWER      <Reviewer's name> <(Reviewer's E-mail address)>
// !REVIEW_DATE <date>
///////////////////////////////////////////////////////////////////////////
class EXPORT utg : public dtCore
{
```

6 7 5 8

```
public:
    // Public enumeration should not end up in inspection class.
    enum stringSize
    {
        STRING_SIZE = 10
    };
```

9 10

```
    ////////////////////////////////////////////////////
    // !METHOD_DECL_S
    // !NAME utg_def_ctor
    // !TEXT
    // The default constructor.  Most classes are
    // required to have one.
    ////////////////////////////////////////////////////

    utg( dtMemoryMgr* aMemMgr );

    ////////////////////////////////////////////////////
    // !METHOD_DECL_S
    // !NAME utg_ctor1
```

11

APPENDIX A

UTG.HH

```
   utg& operator += ( dtInt aRhs);                                        16

   ////////////////////////////////////////////////
   // !METHOD_DECL_S
   // !NAME utg_openForRead
   // !TEXT
   // Static method generates a utg.
   // Static specifier is dropped on inspection
   // member function.
   ////////////////////////////////////////////////                       17

   static erStatus openForRead( utg*& aUtg, dtMemoryMgr* aMM);

   // Public member data.  Should not appear in generated inspection class.  18
   dtUInt       publicID;
   dtChar       publicIDstring[STRING_SIZE];

protected: 19

   ////////////////////////////////////////////////
   // !METHOD_DECL_S
   // !NAME utg_operator[]()
   // !TEXT
   // An overloaded index operator.
   ////////////////////////////////////////////////                       20

   dtChar operator []( dtInt aIndex ) const;

   ////////////////////////////////////////////////
   // !METHOD_DECL_S
   // !NAME utg_rtnID
   // !TEXT
   // Returns ID. A pure virtual member function.
   // makes this an abstract class.  These don't
   // get tested by the unit test, but must be
   // declared (as private) in the inspection class                        21
   // to satisfy compiler (instantiating an abstract
   // class).  No source is generated for this, it
   // can be declared and implemented.
   ////////////////////////////////////////////////

   virtual dtULong rtnID() const = 0;

   ////////////////////////////////////////////////
   // !METHOD_DECL_S
   // !NAME utg_setID
   // !TEXT
   // Set the ID, has an optional parameter.                               22
   // Virtual function declaration is not used
   // on generated inspection member function.
   ////////////////////////////////////////////////

   virtual dtBoolean setID( const dtULong aID,
                const dtChar* const aName = (const dtChar* const) NULL );

#ifdef    OPLDEBUG
   ////////////////////////////////////////////////
   // !METHOD_DECL_S
   // !NAME utg_dump
   // !TEXT
   // This is used by the announceXXX() methods, so                        23
   // there is no need to include this in the
   // generated inspection class.
   ////////////////////////////////////////////////

   dtVoid dump(dtOStream& aOutStream, dtUInt aIndentLevel = 0) const;
```

```
/////////////////////////////////////////////////////////////////////////
//
// FILE_NAME: i_utg.hh 3
//
// Copyright 1993 by Marcam Corp., Newton, MA USA
//
// This unpublished copyrighted work contains
// TRADE SECRET information of Marcam Corporation.
//
// Use, transfer, disclosure, or copying without
// its expressed written permission is strictly
// forbidden.
//
//
/////////////////////////////////////////////////////////////////////////   1

#ifndef i_utgHH
#define i_utgHH   2

#include "idttest.hh"          // For idtTest  4
#include "utg.hh"              // For utg  5

/////////////////////////////////////////////////////////////////////////
// !CLASS_DECL_S
// !LIBRARY I_ER  7
// !NAME i_utg  8
// !TEXT
// Generated 7/12/93 by tutg version 1.0  9
//
// !AUTHOR       <Author's name>   <(Author's E-mail address)>
// !REVIEWER     <Reviewer's name> <(Reviewer's E-mail address)>
// !REVIEW_DATE <date>
/////////////////////////////////////////////////////////////////////////   6


/////////////////////////////////////////////////////////////////////////
// Class under test.  This class is derived from the production object.
// This provides access to the public and protected member functions
// of the production class, without modification to the production
// class.  Also, pure virtual functions declared in the production class
// are provided an implementation here.  This allows production abstract
// base classes to be instantiated and tested.
/////////////////////////////////////////////////////////////////////////

class s_utg : public utg  10
{
public:
    friend class i_utg;  11

    //////////////////////////////////////////////////
    // Inline constructors, and copy constructor
    // for replicating the production class interface.
    //////////////////////////////////////////////////

    s_utg( dtMemoryMgr* aMemMgr ) : utg( aMemMgr ){}

    s_utg( dtULong aArg1,
           const dtChar* const aArg2,
           dtMemoryMgr* aMemMgr ) : utg( aArg1, aArg2, aMemMgr ){}   12

    s_utg( dtULong aArg1,
           const dtChar* const aArg2,
           dtInt aArg3,
           dtMemoryMgr* aMemMgr ) : utg( aArg1, aArg2, aArg3, aMemMgr ){}

    s_utg( const s_utg& aS_utg,
```

I_UTG.HH

```
    s_utg* newInstance( dtMemoryMgr* aMemMgr );

    s_utg* newInstance( dtULong              aArg1,
           const dtChar* const aArg2,
           dtMemoryMgr*        aMemMgr );

    s_utg* newInstance( dtULong              aArg1,
           const dtChar* const aArg2,
           dtInt               aArg3,
           dtMemoryMgr*        aMemMgr );
```
20

```
    //////////////////////////////////////////////////
    // Copy constructor for class under inspection.
    //////////////////////////////////////////////////

    s_utg* copyInstance( s_utg* aInstance,
         dtMemoryMgr* aMemMgr=(dtMemoryMgr*)NULL);
```
21

```
    //////////////////////////////////////////////////
    // Destructor for class under inspection.
    //////////////////////////////////////////////////

    dtVoid deleteInstance( s_utg* aInstance );
```
22

```
    //////////////////////////////////////////////////
    // Other member functions.
    //////////////////////////////////////////////////
```
23

```
    s_utg& oper_equal( s_utg* aInstance,
           const s_utg& aRhs );
```
24

```
    dtULong oper_dtULong( s_utg* aInstance );
```
25

```
    s_utg& oper_plusequal( s_utg* aInstance,
           dtInt aRhs );
```
26

```
    erStatus openForRead( s_utg*& aUtg, dtMemoryMgr* aMM);
```
27

```
    dtChar oper_index( s_utg* aInstance,
           dtInt aIndex );
```
28

```
    dtBoolean setID( s_utg* aInstance,
           const dtULong aID,
           const dtChar* const aName = (const dtChar* const) NULL );
```
29

```
private:
```
30

```
    //////////////////////////////////////////////////
    // Inspection class member functions, unimplemented
    // copy constructor and assignment operator.
    //////////////////////////////////////////////////

    i_utg( const i_utg& aI_utg );

    i_utg& operator=( const i_utg& aI_utg );
```
31

```
};

//////////////////////////////////////////////////
// Nonclass functions to be tested.
//////////////////////////////////////////////////

dtBoolean i_utg_export( idtTest* aTestObj, const s_utg& aUtg );
```
32

```
s_utg& i_utg_oper_plus( idtTest* aTestObj, const s_utg& aLhs, const s_utg& aRhs );
```
33

```
#endif // i_utgHH
```
34

I_UTG.CPP

```
///////////////////////////////////////////////////////////////////////////  2
//
// FILE_NAME: i_utg.cpp
//
// Copyright 1993 by Marcam Corp., Newton, MA USA
//
// This unpublished copyrighted work contains
// TRADE SECRET information of Marcam Corporation.
//                                                                          1
// Use, transfer, disclosure, or copying without
// its expressed written permission is strictly
// forbidden.
//
//
///////////////////////////////////////////////////////////////////////////

#include "i_utg.hh"            // For i_utg  3

// Establish an instance of TESTRUN for this class  4
TESTRUNdeclare(i_utg)

//
// TESTRUN() and testname() are used by the unit test harness
// to run the tests and display the name of the unit test.
//
extern "C" dtInt FAR PASCAL EXPORT testmain( idtTestArgs& aTestArgs )
{
    return( TESTRUN(i_utg)( aTestArgs ) );                                  5
}

extern "C" dtChar * FAR PASCAL EXPORT testname()
{
    return( "i_utg" );
}

///////////////////////////////////////////////////////////////////////////
// !CLASS_DESC
// !LIBRARY ER  7
// !NAME i_utg  8
// !TEXT
// Generated 7/12/93 by utg version 1.0  9
//                                                                          6
// !AUTHOR       <Author's name>   <(Author's E-mail address)>
// !REVIEWER     <Reviewer's name> <(Reviewer's E-mail address)>
// !REVIEW_DATE <date>
///////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////
// Inspection class constructor.
//////////////////////////////////////////////////

i_utg::i_utg(
    dtStreamExecutive* aStreamExec,
    dtMemoryExecutive& aMemExec,
    idtTestArgs&       aTestArgs,                                           10
    dtMemoryMgr*
    ): idtTest( aStreamExec, aMemExec, aTestArgs )
{
    for ( int i = 0 ; i < TOTAL_FUNCS ; i++ )
        coverage[i] = 0;
}

//////////////////////////////////////////////////
// Inspection class destructor.                                             11
```


APPENDIX C

```
18  // has a value after assignment) must be tested.
    //
    TLIFE_START( "ctor1 and operator=" )
        s_utg* testobjX = newInstance( /*PARAMETERS,*/ mm );   19
        s_utg* testobjY = newInstance( /*PARAMETERS,*/ mm2 );   20
        *testobjY = *testobjX;
//!FIX      checkExpr( *testobjX == *testobjY, currTest );
        deleteInstance( testobjX );
        announceObject( *testobjY );
        deleteInstance( testobjY );
21  TLIFE_END

    TLIFE_START( "Chained assignment" )
        s_utg* testobjX = newInstance( /*PARAMETERS,*/ mm );
        s_utg* testobjY = newInstance( /*PARAMETERS,*/ mm2 );   22
        s_utg* testobjZ = newInstance( /*PARAMETERS,*/ mm );
        *testobjZ = *testobjY = *testobjX;
//!FIX      checkExpr( *testobjZ == *testobjY, currTest );
        deleteInstance( testobjX );
        deleteInstance( testobjY );
23      announceObject( *testobjZ );
        deleteInstance( testobjZ );
    TLIFE_END

    TLIFE_START( "Assignment to self" )
        s_utg* testobjX = newInstance( /*PARAMETERS,*/ mm );   24
        *testobjX = *testobjX;
        announceObject( *testobjX );
        deleteInstance( testobjX );
    TLIFE_END

25  TLIFE_CLEANUP;
}

/////////////////////////////////////////////////////
// Operators testing.
//
// All operators are tested here.  The entire
// set of operators has been broken into like
// categories for ease of testing.  Between the          26
// T_INIT and T_CLEANUP macros the memory manager
// variable mm is available for constructing
// test objects.  Each individual test must be
// surrounded by the T_START and T_END macros,
// or alternately, the test itself can be performed
// within a TEST macro.
/////////////////////////////////////////////////////

dtVoid
i_utg::t_Operators()
{
    T_INIT( "t_Operators" )

    // COMPARISON OPERATORS: < > <= >= == !=

    // ARITHMETIC OPERATORS: + - * / % ++ --

    // LOGICAL OPERATORS: || && !

    // BITWISE OPERATORS: ^ | & ~ << >>                     27

    // EXTENDED ASSIGNMENT OPERATORS: += -= *= /= %= ^= |= &= <<= >>=

    // CONVERSION OPERATORS:  type()
```

```
// constructing test objects.  Each individual
// test must be surrounded by the T_START and T_END
// macros, or alternately, the test itself can be
// performed within a TEST macro.
//////////////////////////////////////////////////

dtVoid
i_utg::t_Persist()
{
    T_INIT( "t_Persist" )

    //!FIX Add your persistent member function tests here.


    T_CLEANUP;
}

//////////////////////////////////////////////////
// Return coverage information.
//
// NOTE: This is not to be modified by unit
//       test developers.
//////////////////////////////////////////////////

dtVoid
i_utg::qryCoverage(
    dtInt*& aCovAry,
    dtInt&  aNumFuncs
    ) const
{
    aCovAry = coverage;
    aNumFuncs = TOTAL_FUNCS;
}

//////////////////////////////////////////////////
// Constructors for class under inspection.
//
// NOTE: The following constructors can
//       be modified by unit test developers.
//       The parameters for a given constructor,
//       and the announcement of parameters and
//       resulting objects can be customized
//       by the unit test developer.
//////////////////////////////////////////////////

s_utg*
i_utg::newInstance(
    dtMemoryMgr* aMemMgr
    )
{
    coverage[f_newInstance1]++;
    announceMethodStart( "newInstance1" );
    s_utg* NewObj = new (aMemMgr) s_utg(aMemMgr);
    announceMethodEnd( "newInstance1" );
    return( NewObj );
}

s_utg*
i_utg::newInstance(
    dtULong              aArg1,
    const dtChar* const  aArg2,
    dtMemoryMgr*         aMemMgr )
{
    coverage[f_newInstance2]++;
    announceMethodStart( "newInstance2" );
    s_utg* NewObj = new (aMemMgr) s_utg( aArg1, aArg2, aMemMgr );
```

30

31

32

33

34

35

36

37

38

```
    coverage[f_oper_ qual]++;
    announceMethodStart( "oper_equal" );
    announceParameter( aRhs );
    aInstance->operator=( *aInstance ); —42
    announceMethodEnd( "oper_equal" );
    return( *aInstance ); 43
}


dtULong
i_utg::oper_dtULong(
    s_utg* aInstance
    )
{
    coverage[f_oper_dtULong]++;
    announceMethodStart( "oper_dtULong" );
    dtULong testVal = aInstance->operator dtULong(); — 44
    announceMethodEnd( "oper_dtULong" );
    return( testVal ); 45
}

s_utg&
i_utg::oper_plusequal(
    s_utg* aInstance,
    dtInt aRhs
    )
{
    coverage[f_oper_plusequal]++;
    announceMethodStart( "oper_plusequal" );
//    announceParameter( aRhs );
    aInstance->operator+=( *aInstance );
    announceMethodEnd( "oper_plusequal" );
    return( *aInstance );
}


dtChar
i_utg::oper_index(
    s_utg* aInstance,
    dtInt aIndex
    )
{
    coverage[f_oper_index]++;
    announceMethodStart( "oper_index" );
//  announceParameter( aIndex );
    dtChar testVal = aInstance->operator[](aIndex);
    announceMethodEnd( "oper_index" );
    return( testVal );
}


dtBoolean
i_utg::setID(
    s_utg* aInstance,
    const dtULong aID,
    const dtChar* const aName
    )
{
    coverage[f_setID]++;
    announceMethodStart( "setID" );
//  announceParameter( aID );
//  announceParameter( aName );
    dtBoolean testVal = aInstance->setID(aID,aName);
    announceMethodEnd( "setID" );
    return( testVal );
}
```

```
////////////////////////////////////////////////////////////////////////
//
// FILE_NAME:   basresc.hh
//
// Copyright 1993 by Marcam Corp., Newton, MA USA
//
// This unpublished copyrighted work contains
// TRADE SECRET information of Marcam Corporation.
//
// Use, transfer, disclosure, or copying without its expressed
// written permission is strictly forbidden.
//
////////////////////////////////////////////////////////////////////////

#ifndef basrescHH
#define basrescHH

#include "geninst.hh"        // For fdGenInst
#include "bcdatet.hh"        // For bcDateTime
#include "nameuk.hh"         // For fdNameUK
#include "dtbitv.hh"         // For dtBitVector (dummy)

class EXPORT BasicResource;

#ifdef __OSE_TEMPLATES__
#include "orb.hh"
#endif

#ifndef SCHEMACOMPILER
#include "basresc.h"
#endif

////////////////////////////////////////////////////////////////////////
// !CLASS_DECL_S
// !LIBRARY
// !NAME BasicResource
// !TEXT
// BasicRecource is the implementation of a generic instance-derived
// class.  This is an independently persistable kind (IPK).
//
// The user's access rights will be checked whenever fdSecurityContext
// appears as a parameter.  If the user does not have appropriate access
// in the standard UI/App API, an error message will be returned.
//
// A returned pointer to an erMsg or erMsgList indicates an error
// occurred.  A NULL pointer indicates successful completion of
// the member function.
//
// !AUTHOR       <Author's name>   <(Author's E-mail address)>
// !REVIEWER     <Reviewer's name> <(Reviewer's E-mail address)>
// !REVIEW_DATE <date>
//
////////////////////////////////////////////////////////////////////////


class EXPORT BasicResource : public fdGenInst
{
  public:

    ////////////////////////////////////////////////////
    // !METHOD_DECL_S
    // !NAME close
    // !TEXT
    // This static method closes the current object,
    // releasing all database and memory resources.
    // When successful, the aBasicResource pointer
```

BASRESC.HH

```
////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME openForReview
// !TEXT
// This static method retrieves an existing BasicResource
// from the database by its user key in a nonmodifiable
// mode. If an error occurs the aBasicResource
// pointer will be set to Null, and an error message
// is returned.
//
// **Standard UI/App API, implemented in stdgi.cc**
////////////////////////////////////////////////

static erMsg* openForReview( BasicResource*&     aBasicResource,
                             const fdNameUK&     aBasicResourceUK,
                             dbTransCtx*&        a...tx,
                             fdSecurityContext   aSe...tyCtx,
                             fdInstTypes::type   aType,
                             dtMemoryMgr*        aMM );

////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME remove
// !TEXT
// This static method removes an existing BasicReso...e
// from the database. When successful, the
// aBasicResource pointer is Null (the object has been
// deleted from memory). If an error occurs, e.g.,
// validateToRemove() fails, then an error message
// is returned.
//
// **Standard UI/App API, implemented in stdgi.cc**
////////////////////////////////////////////////

static erMsgList* remove( BasicResource*&     aBasicResource,
                          dbTransCtx*&        aDbCtx,
                          fdSecurityContext   aSecurityCtx );

////////////////////////////////////////////////
// !M...D_DECL_S
// !... saveAs
// !...
// ... static method saves the current object as a new
// object using the supplied user key and type. The
// original object (aOrigBasicResource) is closed,
// and the new object (aNewBasicResource) is saved
// to the database and remains in memory. Validation
// errors during save are reported.
//
// **Standard UI/App API, implemented in stdgi.cc**
////////////////////////////////////////////////

static erMsgList* saveAs( BasicResource*&     aNewBasicResource,
                          const fdNameUK&     aBasicResourceUK,
                          BasicResource*&     aOrigBasicResource,
                          dbTransCtx*&        aDbCtx,
                          fdSecurityContext   aSecurityCtx,
                          fdInstTypes::type   aType,
                          dtMemoryMgr*        aMM );

////////////////////////////////////////////////
// !METHOD_DECL_S
// !NAME validateToRemove
// !TEXT
// This virtual method (declared pure virtual by fdGenInst)
// performs all validation (referential integrity)
// required before removing this object from the database.
```

USER KEY
CLASS NAME

I_BASRES.CC

```
/////////////////////////////////////////////////////
// Persistence testing.
//
// This tests behavior specific to the persistence
// member functions of a production class.
//
// Between the T_INIT and T_CLEANUP macros the
// memory manager variable mm is available for
// constructing test objects.  Each individual
// test must be surrounded by the TEST_START and TEST_END
// macros, or alternately, the test itself can be
// performed within a TEST macro.
/////////////////////////////////////////////////////

dtVoid
i_BasicResource::t_Persist()
{
    // STDGIT_TESTCLASS and STDGIT_OBJREF must be established
    //
    #define STDGIT_TESTCLASS s_BasicResource
    #define STDGIT_OBJREF    oObjRef<BasicResource>

    // At least one of the following must be present.
    //
    #define STDGIT_PRODUCTION
    //#define STDGIT_WIP
    //#define STDGIT_TEMPLATE
    //#define STDGIT_IMPORT_EXPORT

    T_INIT( "t_Persist" )

    fdNameUK uk1( mm );
    fdNameUK uk2( mm );
    fdNameUK uk3( mm );
    STDGIT_TESTCLASS* obj1 = (STDGIT_TESTCLASS*) NULL;
    STDGIT_TESTCLASS* obj2 = (STDGIT_TESTCLASS*) NULL;
    STDGIT_TESTCLASS* obj3 = (STDGIT_TESTCLASS*) NULL;
//!FIX  You may need to specify a security context.
    fdSecurityContext security = FD_SC_NO_CONTEXT;
//!FIX  When implemented by STDGIT.CC the KindRef required is named "kr".
//!FIX  fdKindRef< oObjRef<BasicResource>, fdNameUK > kr;

    // Setup the user keys for the 3 test objects.
    //
    setupUKs( uk1, uk2, uk3, mm );

    // Standard test for fdGenInst derived classes.
    // Runs through normal persistent lifecycle for
    // production instances.
    //
    #include "stdgit.cc"

    T_CLEANUP
}

/////////////////////////////////////////////////////
// Return coverage information.
//
// NOTE: This is not to be modified by unit
//       test developers.
/////////////////////////////////////////////////////
dtVoid
i_BasicResource::qryCoverage(
  dtInt*& aCovAry,
  dtInt&  aNumFuncs
  ) const
{
  aCovAry = coverage;
```

11/8/93 1:12PM Page 1

APPENDIX F

```
//
// Test 1: First Constructor & Destructor
//

    // Set up test...
TLIFE_START( "Ctor1 and Dtor" )                Memory manager mm set up, scope established

①       // First, run the constructor, then dump the object, then delete it.
    s_erMsg* msg1 = newInstance( fdMsgTest, ERMSGTEST_NAME, ERMSGTEST_1, __FILE__, __LINE__, -- );
②   announceObject( *msg1 );                                        Standard erMsg Parameters
③   deleteInstance( msg1 );

    // Complete Lifecycle test...
TLIFE_END                                      Scope ended, Memory manager checked for leaks
```

APPENDIX G

```
//
// Test 1: qryMsgID
//
s_erMsg* msg1 = newInstance( idMsgTest3, ERMSGTEST_NAME, ERMSGTEST_3, __FILE__, __LINE__ );
TEST( "qryMsgID", idMsgTest3 == qryMsgID( msg1 ) )
```

APPENDIX I

```
TESTRUNdeclare(erMsg1)

extern "C" {
        dtChar * FAR PASCAL EXPORT testname()
    {
        return( "erMsg1" );
    }

    dtInt FAR PASCAL EXPORT testmain( idtTestArgs& aTestArgs )
    {
    return( TESTRUN(erMsg1)( aTestArgs ) );
    }
}       // end extern C
```

APPENDIX K

```
****************************************
 Starting test: Copy Ctor - Copying Ctor1 across Memory Mgrs, at line: 133, File: c:\view\prodn\fd\...
ERROR: MEMORY LEAK DETECTED IN MEMORY MANAGER, in test: Copy Ctor - Copying Ctor1 across Memory Mgrs, at Line: 146
ERROR: MEMORY LEAK DETECTED IN MEMORY MANAGER, in test: Copy Ctor - Copying Ctor1 across Memory Mgrs, at Line: 146
 FAILED test: Copy Ctor - Copying Ctor1 across Memory Mgrs, at line: 146, File: c:\view\prodn\fd\...
****************************************
...
****************************************
Starting test: qryMsgID, at line: 268, File: c:\view\prodn\fd\errors\ut\insp\erMsgi.cpp
 CHECKED: Expression: qryMsgID( msg1 ) == 42, in test: qryMsgID, at line: 268, File: c:\view\prodn\fd\...
 PASS test: qryMsgID, at line: 268, File: c:\view\prodn\fd\errors\ut\insp\erMsgi.cpp
****************************************
```

APPENDIX M

CLAIMS:

1. A method for testing a subject class in an object-oriented digital data processing system, said method comprising the step of:

(A) responding to a source signal defining said subject class to generate an inspection signal defining an inspection class having one or more members for

(i) creating a test object as an instantiation of any of said subject class and a class derived therefrom,

(ii) invoking one or more selected method members of said test object,

(iii) generating a signal, hereinafter referred to as a report signal, reporting an effect of such invocation.

2. A method according to claim 1, comprising

(B) responding to said source signal to generate a test signal defining a test class that comprises any of

(i) said subject class, and

(ii) an instantiable class derived from said subject class, and

step (A)(i) includes the step of generating said inspection signal to define said inspection class to include one or more members for creating said test object as an instantiation of said test class.

3. A method according to claim 2, wherein

step (B) includes the step of generating said test signal to define an instantiable test class that inherits one or more members of said subject class.

4. A method according to claim 3, wherein

step (B) includes the step of generating said test signal to define said test class to substantially duplicate pure virtual functions of said subject class, absent constructs that denote those functions as having both pure and virtual attributes.

5. A method according to claim 3, wherein

step (B) includes the step of generating said test signal to define said test class to substantially duplicate at least one constructor and a destructor of said subject class.

6. A method according to claim 3, wherein

step (A) includes the step of generating said inspection signal to define said inspection class to comprise one or more method members providing common reporting services for invocation by said test suite members.

13. A method according to claim 7, wherein

step (A) includes the step of generating said inspection signal to define said inspection members to include method member functions corresponding to at least one constructor in said test object, wherein each of those method member functions take substantially the same arguments as the corresponding constructor in said test object.

14. A method according to claim 7, wherein

step (A) includes the step of generating said inspection signal to define said inspection members to include method member functions corresponding to a destructor in said test object.

15. A method according to claim 7, wherein

step (A) includes the step of generating said inspection signal to define said inspection members to include method member functions corresponding to at least one operator function in said test object.

16. A method according to claim 7, wherein

step (A) includes the step of generating said inspection signal to define said inspection members to have function names similar to those of the corresponding method members of the test object that they test.

17. A method according to claim 1, comprising the step of

(B) responding to said inspection signal to create an inspection object as an instantiation of said inspection class, and

(C) invoking one or more members of said inspection object to

(i) create said test object,

(ii) invoke one or more selected method members of said test object,

(iii) generate a signal, hereinafter referred to as a report signal, reporting an effect of such invocation.

(i) a comparison of results of invocation of one or more method members of said test object with one or more expected values thereof, and

(ii) detection of a memory leak.

26. A method according to claim 17, wherein

said method includes the step of executing step (B) a plurality of times, each for invoking a different group of one or more selected method members of said test object.

27. A method according to claim 17, wherein

step (C)(ii) includes the step of invoking one or more method members of said inspection object, referred to hereinafter as inspection members, for testing corresponding method members of said test object.

28. A method according to claim 27, wherein

step (C)(ii) includes the step of invoking one or more method members one or more members of said inspection object, referred to hereinafter as test suite members, for exercising one or more of said inspection members.

29. A method according to claim 28, wherein

step (C)(ii) includes the step of invoking one or more test suite members to at least one of

(i) test for memory leaks in connection with at least one of creation and destruction of said test object,

(ii) test accessor and transformer members of said test object,

(iii) test operator member methods of said test object,

(iv) test members involved in persistence of said test object, and

(v) test method members semantically unique to said test object.

30. A method according to claim 28, wherein

step (C)(ii) includes the step invoking one or more method members of said inspection object for invoking one or more test suite members.

31. A method according to claim 17, wherein

36. A method according to claim 34, wherein

step (B)(ii) includes applying at least one argument to one or more of said selected method members of said test object in connection with invocation thereof.

37. A method according to claim 36, wherein

step (B)(ii) includes applying at least one argument to a constructor member of said test object in connection with creation of said test object.

38. A method according to claim 34, wherein

step (B)(ii) includes the step of preventing exceptions that occur during invocation of said one or more selected method members from discontinuing execution of steps (B)(ii) and (B)(iii).

39. A method according to claim 34, wherein

step (B)(iii) includes comparing said result of such invocation with one or more expected values thereof, and generating said report signal to be indicative of such comparison.

40. A method according to claim 39, wherein

step (B)(iii) includes the step of storing a signal indicative of said effects in a data member of said inspection object.

41. A method according to claim 34, wherein

said method includes the step of executing step (B) a plurality of times, each for invoking a different group of one or more selected method members of said test object.

42. A method according to claim 41, wherein

step (B)(iii) includes the step of generating said report signal to be indicative of at least one of

(i) comparison of results of invocation of one or more method members of said test object with one or more expected values thereof, and

(ii) detection of a memory leak.

43. A method according to claim 34, wherein

responding to a verbosity control signal to generate said report signal with a selected level of verbosity.

50. Apparatus for testing a subject class in an object-oriented digital data processing system, said apparatus comprising:

(A) code generator means for responding to a source signal defining said subject class to generate an inspection signal defining an inspection class having one or more members for

(i) creating a test object as an instantiation of any of said subject class and a class derived therefrom,

(ii) invoking one or more selected method members of said test object,

(iii) generate a signal, hereinafter referred to as a report signal, reporting an effect of such invocation, and

(B) test harness means, coupled to said code generator means, for responding to said inspection signal for creating an inspection object instantiating said inspection class, and for generating an inspection object invocation signal for invoking one or more members thereof.

51. An apparatus according to claim 50, wherein

said code generator means includes test class generating means for responding to said source signal to generate a test signal defining a test class that comprises any of

(i) said subject class, and

(ii) an instantiable class derived from said subject class, and wherein

said code generator further includes means for generating said inspection signal to define said inspection class to include one or more members for creating said test object as an instantiation of said test class.

52. An apparatus according to claim 50, wherein

said code generator means includes means for generating said inspection signal to define said inspection class to include one or more method members, referred to hereinafter as inspection members, for testing corresponding method members of said test object.

53. An apparatus according to claim 52, wherein

(a) said subject class. and

(b) an instantiable class derived from said subject class.

59. An apparatus according to claim 57, wherein

said inspection object execution means includes means for applying at least one argument to one or more of said selected method members of said test object in connection with invocation thereof.

60. An apparatus according to claim 57. wherein

said inspection object execution means includes exception service means preventing exceptions that occur during invocation of said one or more selected method members from discontinuing at least reporting on effects of invocation of said test object.

61. An apparatus according to claim 57. wherein

said inspection object execution means includes means for comparing a result of invocation of said one or more selected method members of said test object with expected results of such invocation, and for generating said report signal to be indicative of such comparison.

62. An apparatus according to any of claims 57 and 59. wherein

said inspection object execution means includes means for executing one or more member methods of said inspection object to generate said report signal.

63. An apparatus according to any of claims 61. wherein:

said inspection object execution means includes means responding to a verbosity control signal to generate said report signals with a selected level of verbosity.

64. An apparatus according to claim 57. wherein

said inspection object execution means includes means for executing one or more method members of said inspection object, referred to hereinafter as inspection members. for testing corresponding method members of said test object.

65. An apparatus according to claim 64, wherein

for creating an inspection object instantiating an inspection class. and for generating an inspection object invocation signal for invoking one or members thereof.

(B) inspection object execution means, coupled to said test harness means. for responding to said inspection object invocation signal for

(i) creating said test object.

(ii) generating a test object invocation signal to invoke one or more selected method members of said test object.

(iii) generate a signal, hereinafter referred to as a report signal. reporting an effect of such invocation.

(C) test object execution means. coupled to said inspection object execution means. for responding to said test object invocation signal to execute one or more selected method members thereof.

70. An apparatus according to claim 69. wherein

said inspection object execution means includes means for creating said test object as an instantiation of a test class that comprises any of

(a) said subject class. and

(b) an instantiable class derived from said subject class.

71. An apparatus according to claim 69. wherein

said inspection object execution means includes means for applying at least one argument to one or more of said selected method members of said test object in connection with invocation thereof.

72. An apparatus according to claim 69. wherein

said inspection object execution means includes exception service means preventing exceptions that occur during invocation of said one or more selected method members from discontinuing at least reporting on effects of invocation of said test object.

73. An apparatus according to claim 69. wherein

said inspection object execution means includes means for comparing a result of invocation of said one or more selected method members of said test object with expected

said inspection object execution means includes means for executing a method member of said inspection object for invoking one or more test suite members.

80. An apparatus according to claim 69, wherein

said inspection object execution means includes means for placing said digital data processor in a desired runtime environment.

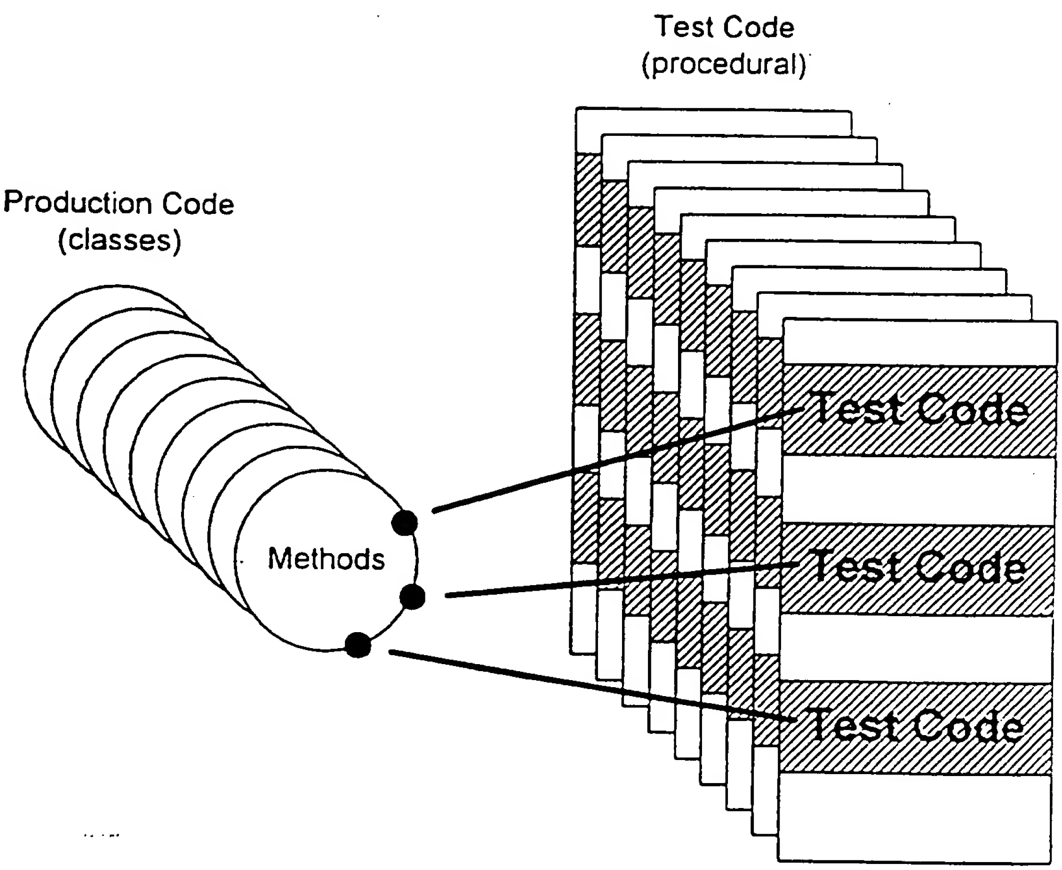


Figure 2
(prior art)

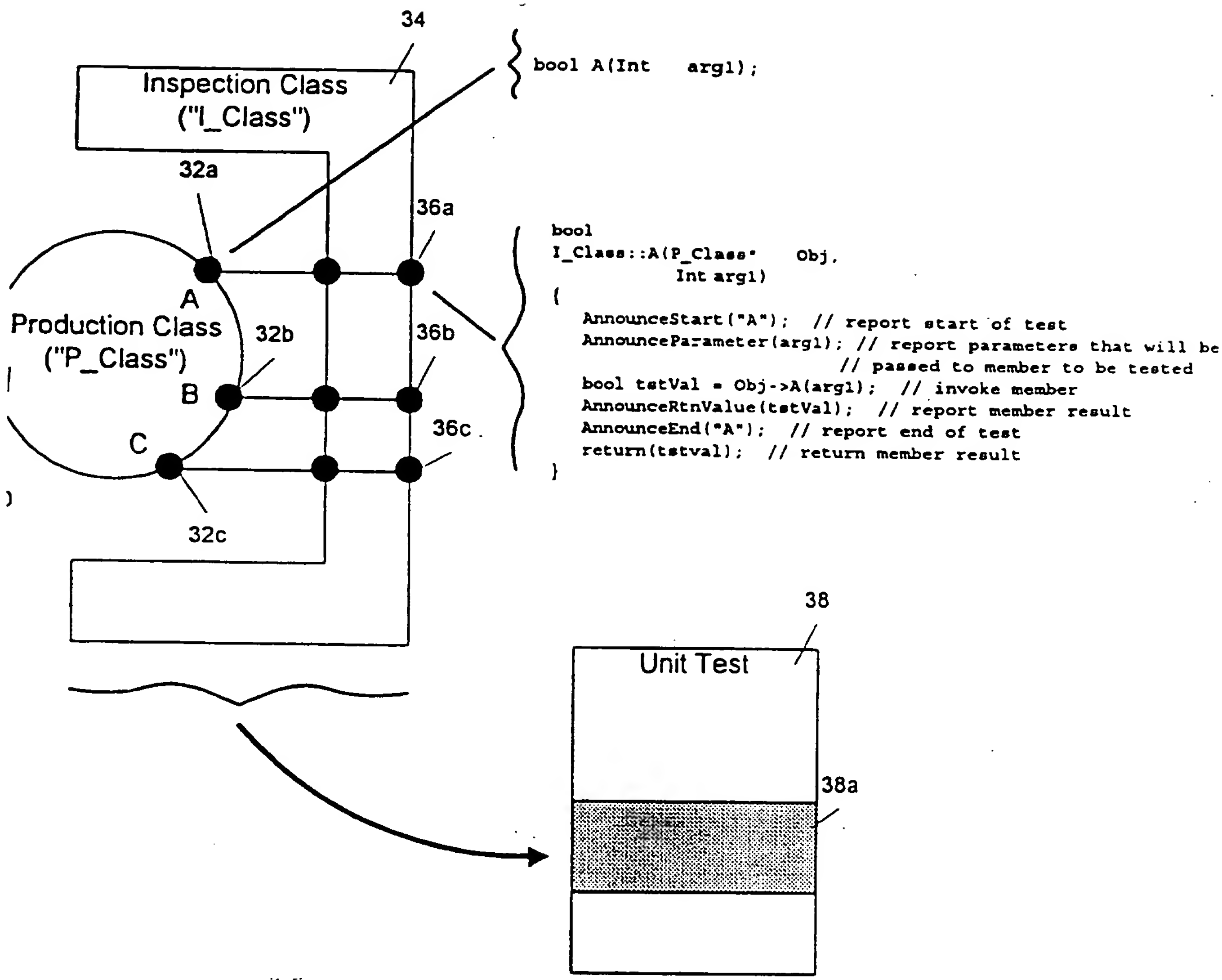

34
Inspection Class
("I_Class")
bool A(Int arg1);
32a
36a
A
Production Class
("P_Class")
32b
36b
B
C
36c
32c
bool
I_Class::A(P_Class* Obj,
Int arg1)
{
AnnounceStart("A"); // report start of test
AnnounceParameter(arg1); // report parameters that will be
// passed to member to be tested
bool tstVal = Obj->A(arg1); // invoke member
AnnounceRtnValue(tstVal); // report member result
AnnounceEnd("A"); // report end of test
return(tstval); // return member result
}
38
Unit Test
38a

Figure 4

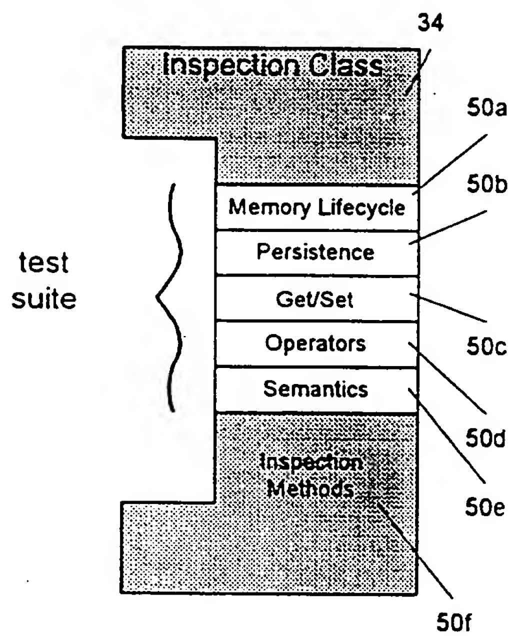

34
Inspection Class
50a
50b
Memory Lifecycle
Persistence
test suite
Get/Set
50c
Operators
Semantics
50d
Inspection Methods
50e
50f

Figure 6

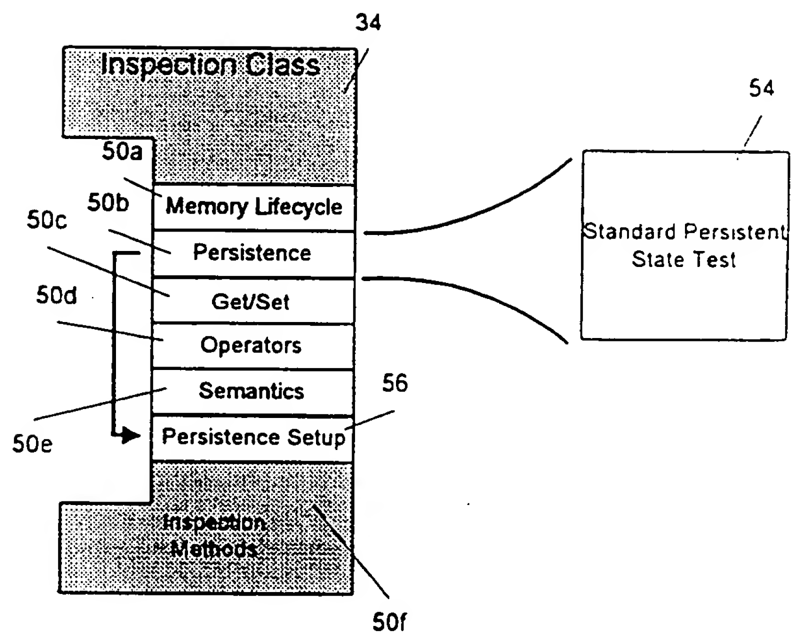

34
Inspection Class
54
50a
Memory Lifecycle
50b
50c
Persistence
Standard Persistent State Test
Get/Set
50d
Operators
Semantics
56
50e
Persistence Setup
Inspection Methods
50f

Figure 8

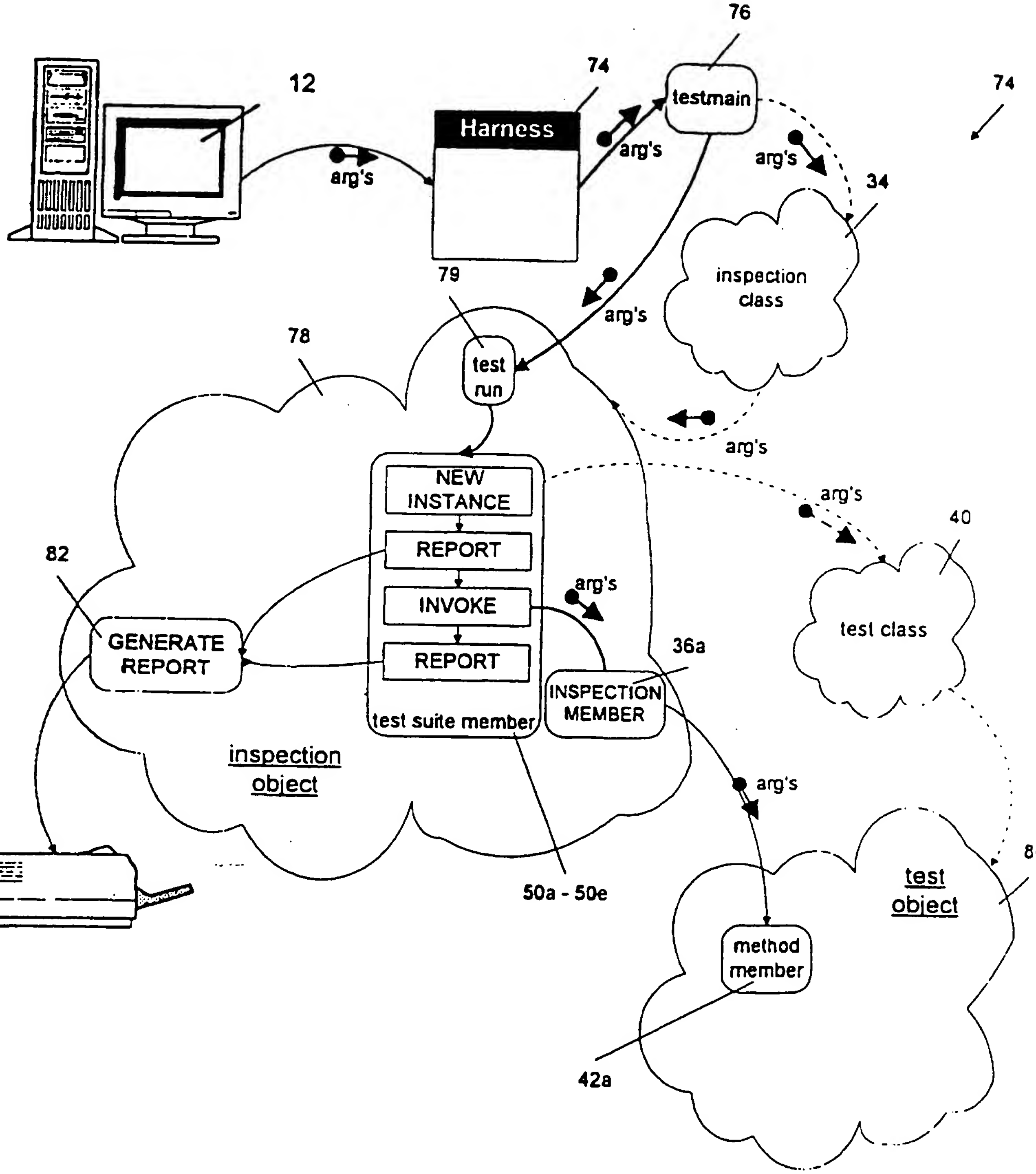

12
74
76
74
Harness
testmain
arg's
arg's
arg's
34
inspection class
79
78
arg's
test run
arg's
NEW INSTANCE
REPORT
INVOKE
REPORT
test suite member
arg's
40
82
GENERATE REPORT
arg's
test class
36a
INSPECTION MEMBER
inspection object
arg's
25
80
test object
50a - 50e
method member
42a

Figure 10

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US95/06059

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :G06F 9/445,9/45
US CL :395/700
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/700, 364/275.5

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

IEEE online

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	Meyer, Object-Oriented Software Construction, Prentise-Hall, 1988, pages 65-104, and especially pages 347-348.	1-80
X	US, A, 5,093,914 (Coplien, et al.) 03 March 1992, col. 1-26.	1-80
Y	US, A, 4,885,717 (Beck, et al.), 05 December 1989, col. 1-16.	1-80
Y	US, A, 4,989,132 (Mellander, et al.), 29 January 1991, col. 46-54.	1-80

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be part of particular relevance

"E" earlier document published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
04 SEPTEMBER 1995	22 SEP 1995

Name and mailing address of the ISA/US	Authorized officer
Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231	B. Harden for JON BACKENSTOSE
Facsimile No. (703) 305-3230	Telephone No. (703) 305-9661